Values and References

One can view a variable as a box that contains some object. It has a name, and a place of its own where it can store an object.

A reference has a name, but no place of its own. It uses the storage place of another variable. The other variable has to be specified when the reference is created.

References

A reference is recognized by `&`. If T is a type, then T`&` is the type of references over T.

```
int x;
int& y = x;

x = 4;
y = 3;
   // What is the value of x?
```

References

```
int p [10];
for( unsigned int i = 0; i < 10; ++ i )
{
   int& x = p[i];
   x = 2 * x;
}
```

Dangers of References

The main problem with references is that: Two variables that appear to be different, are in fact the same.

One sees this in the previous example. Reference x appears to be distinct from array p, but in reality, modifying x means modifying p.

Frame Problem

```
void asktwointegers( int& i1, int& i2 )
{
   std::cout << "please type two integers: ";
   std::cin >> i1;
   std::cin >> i2;

   std::cout << "you typed ";
   std::cout << i1 << ", " << i2 << "\n";
}
```

Frame Problem (2)

```
int i, j;

asktwointegers( i, j ); // Fine.
asktwointegers( i, i ); // Does not work.
```

Frame Problem (3)

In languages with lazy semantics (like Java, $C^{\#}$ or Python), the problem is much worse. But it still exists in C^{++}.

The frame problem is: We have to variables $V1$ and $V2$. We change something in $V1$, but suddenly $V2$ also changes.

It is caused by the fact that, due to references or pointers, $V1$ and $V2$ share structure.

Frame Problem (4)

```
addmatrices( matrix& m1, matrix& m2, matrix& m3 );
   // Probably safe.
multmatrices( matrix& m1, matrix& m2, matrix& m3 );
   // Calculating m = p * m is risky.
```

References (and pointers) are dangerous, but they are necessary.

References $\Rightarrow$ Use **const** as much as possible. Pointers $\Rightarrow$ Hide them in classes, and use iterators as much as possible.

Const References

A const reference is a reference s.t. the value of the reference cannot be changed through the reference. It has form const T&.

```
{
   int i = 4;
   const int& j = i;

   std::cout << ( j + j ) << "\n";  // Fine;
   j = 1;                           // Forbidden.
   std::cin >> j;                   // Forbidden.

   i = i + 1;                       // Allowed, and still changes
                                    // the value of j.

}
```

Const References (2)

```
{
   int i = 4;
   const int& j = i;

   int& k = j;     // Not allowed.
                   // Compiler guards constness.
}
```

Const References (3)

Const references can be initialized by value expressions. In that case, the value is guaranteed to exist as long as the reference exists.

```
{
   std::cin >> i;
   const int& j = i + 1;
      // Allowed. (i+1) is guaranteed to
      // exist as long as j exists.

   std::cout << j << "\n";

   int& k = i + 1;
      // Not allowed.
}
```

Importance of References

References are essential for compiling imperative languages. *C* compilers use them internally, in *C*++ they were made explicit.

```
p. x = q. x + 1;
```

A variable reference is always a reference, until the compiler encounters an operation that forces it to make a value out of it.

(`.x` means adding a displacement to the address and changing the type.)

Parameters of Procedures

A parameter of a function or procedure can be either

- A value T. In that case, the procedure receives its own local copy, which it can modify. The modifications have no effect in the calling environment.
- A reference $T\&$. The procedure does not have a local copy. Modifying the parameters means modifying the corresponding variable in the calling environment.
- A const reference const$T\&$. The procedure does not have a local copy, but it cannot modify the corresponding variable in the calling environment.

It is important to understand that in all 3 cases, parameter passing is not special. It is one of the forms of initialization that you have seen already.

Const and Value

There also exist const value variables, but I never used them:

```
const pi = 3.1415926535;
```

(Const value variable are usually static.)

Reference Parameters

Use parameter passing by reference, when

- The function must be able to modify the value its parameter. In general, this should be avoided. It is better to use a function, or to make the function as class method. A class method can return two values, it can return a value and change its class elements.

  If that is not enough, you need reference parameters.

- The object is so big, that you want to avoid copying it. In that case, use const reference.

  (But here, you can also try to make the method a class method.)

When to Use References

There are two reasons for using a reference:

1. References enable a procedure to have output parameters. Most of the time, it is possible to define a function, but not always.

   There may be more more than one output parameter, sometimes it is not natural to use a function.

2. Input parameters that are so big that copying is inefficient.

Safe Usage of References

Use **const** references wherever possible.

Inside a procedure, do not make assumptions about distinctness of references.

If the parameters have different types, you can assume they are disinct:

```
void noproblem1( double& x, unsigned int &i )
   // Safe to assume that x and i are distinct.


double force( double& m, double& a )
   // m is mass, a is acceleration.
   // This is almost the same as m,a having different
   // types. It is safe to assume they are distinct.
```

Danger of References

In the following example, one would like to avoid copying the matrix:

```
matrixmultiply( const matrix& m1, const matrix& m2,
                matrix& prod )
{
   for( unsigned int i = 0; i < 3; ++ i )
       for( unsigned int j = 0; j < 3; ++ j )
          prod [i] [j] = .... ;
}
```

This code will fail if the user writes

```
matrixmultiply( m, other_matrix, m );
```

Pointers and References

Pointers and references are similar things.

- Pointers are intended for the construction of complicated data structures, like trees, arrays or lists.
- Pointers can be used for allocating data structures which have a size that is not known in advance. (vectors, bigints)

For other uses, (parameter passing, abbreviation of long expresssions), use references.

Pointers and References

A reference cannot be distinguished from a value variable.

A pointer can be distinguished.

Operations on Pointers

```
unsigned int *p;
   // Declares, but does not initialize a pointer.
std::cout << *p << "\n";
   // If p points to something, then *p retrieves this
   // something.
p = p + 1;
   // If p is part of an array of integers, then this
   // makes p point to the next array of integers.
++ p;
   // Same as p = p + 1;
```

```
(*p) ++ ;
   // This increases the object that p points to by 1.
std::cout << * ( p + 2 );
   // If p is part of an array of integers, then the
   // value stored 2 positions behind p is printed.
std::cout << p[2] << "\n";
   // Alternative syntax for *(p+2).
```

Allocation and Deallocation

```
int* p = 0;
   // Null pointer. 0 is the value that is guaranteed
   // to point to nothing.

p = new int(4);
   // Creates new int on heap, initializes it with 4.
std::cout << "this is the number four:" << *p << "\n";
*p = 3;
std::cout << "this is the number three:" << *p << "\n";

delete p;
   // Returns the memory to the system.
```

```
// An example of a situation where the size is not
// known in advance:

std::cout << "How many numbers do you need? ";
std::cin >> n;

int* p = new int[n];

for( unsigned int i = 0; i < n; ++ n )
{
   std::cout << "please type " << n << "-th number";
   std::cin >> p[i];
}
```

```
std::cout << "here are the numbers that you typed: ";
for( unsigned int i = 0; i < n; ++ i )
{
   std::cout << i << ": " << p[i] << "\n";
       // p[i] is an abbreviation for *(p+i).
}

delete p;
   // Only pointers that are constructed by
   // new can be deleted.
```

Dangers of Pointers (1)

The frame problem (as with references).

Pointers can be unitialized, point to a position outside of the program's memory, or to nonsense.

Writing to an ill-defined pointer has unpredictable effect.

Deleting data belonging to a pointer that was not constructed by **new** has unpredictable effect. Deleting same pointer twice has unpredictable effect.

Forgetting to delete data causes memory leak.

Dangers of Pointers (2)

In my view, the frame problem with pointers is less dangerous than the frame problem with references:

- Pointers do not look like usual variables.
- Pointers are not used for parameter passing.
- Pointers are used more rarely, only in low level data structures.

Detecting Memory Leaks

Most memory leaks can be detected using **top** command.

Write a procedure of the following form:

```
for( unsigned int k = 0; k < 1000000; ++ k )
{
   suspected_code( );
}
```

Safe Usage of Pointers

You should use pointers only inside the definition of data types, that are big in size or complex in structure.

You should always build your data structures in such a way that the pointers are invisible from the outside, and that the data structures can be used in direct variables.

Comparison with Java and the like

Because in Java (and Python and Ruby) variables implicitly are references, it is impossible to write programs that have direct mathematical semantics in such languages.

For example, if one has a List of Objects in Java, and takes out one element, then it is possible to change the list, by changing the object that was taken out.

In C^{++}, one can build a lot of mess, but if you use the language in the proper way, it is possible to define data structures with precise, functional semantics.

Pure Object-Oriented Programming

The notion 'object' is a word that refers to physical reality. The first object-oriented languages were designed with the purpose of modelling physical entities.

The notion of reference makes sense in the physical world, but no sense in the mathematical universe.

C^{++} is not a purely object-oriented programming language, because not everything is an object, and the programming modal is not state-oriented, but (in my view) it took the good things from object-oriented programming, and left out the rest.

(The good thing is to allow type construction through characteristic functions, instead of only by tuples)

Some Examples

I will show more examples of allocation:

- Single linked lists.
- Strings.
- Binary search trees.

You can obtain functional semantics if you ensure that each allocated entity is reached in only one way.